VERIFICATION OF SYSTOLIC ARRAYS:
A stream functional approach

85-001

SANJAY RAJOPADHYE
AND
PRAKASH PANANGADEN

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF UTAH

UUCS-85-001

VERIFICATION OF SYSTOLIC ARRAYS:

A stream functional approach

Sanjay Rajopadhye
Department of Computer Science
University of Utah
Salt Lake City, Ut 84112

Ph: (801)581-8377
Net Address: sanjay@utah-20

Prakash Panangaden
Department of Computer Science
Cornell University
Ithaca, NY 14853

Ph:(607)256-4052
Net Address: prakash@cornell

Abstract

We illustrate that the verification of systolic architectures can be carried out using techniques developed in the context of verification of programs. This is achieved by a decomposition of the original problem into separately proving the correctness of the data representation and of the individual processing elements in the systolic architecture. By expressing a processing element as a function on a stream of data we are able to utilize standard proof techniques from programming language theory. This decomposition leads to relatively straightforward proofs of the properties of the systolic architecture. We illustrate the techniques via a substantial example, the proof of the correctness of a linear-time systolic architecture for computing the gcd of polynomials. Although this architecture has been designed a few years ago, a formal proof of correctness has not hitherto appeared in the literature.

This research was sponsored in part by the Office of the Naval Research under contract N00014-83-K-0317, and in part by a University of Utah fellowship

I Introduction

In the last few years there has been much interest in the use of formal techniques for the design and analysis of VLSI circuits. Special formalisms have been invented for expressing transformations and specifications of such circuits. One important aspect of the analysis of circuits is the correctness problem. In this paper we show how the correctness problem for an important class of parallel architectures, namely systolic arrays can be decomposed in such a way that formal techniques already developed for reasoning about programming languages can be profitably applied. The particular techniques we use were invented in the context of semantics of functional languages [4] and proving the correctness of data representations [3].

The decomposition consists of separating the *representation* of the original data-domain from the *computational* actions of the circuit on the data elements. Two important features of systolic arrays permit us to reason about this decomposition. Firstly, it is well known that the individual processors in systolic arrays act synchronously[1] on streams of data tokens. Because of this synchronization the processors can be treated as *Kahn processes*. Such processes accept data tokens on their input channels and send tokens on their output channels *without ever polling the channels for availability of data tokens*. Furthermore, by using streams of tokens as the representation data-domain we can use the results of Kahn [4] to eliminate local history in the individual processes and view them as *stream functions*. The action of the entire network can now be viewed as merely the composition of these individual stream functions. We illustrate this technique by proving the correctness of a systolic architecture designed by Brent and Kung [1].

In [1] a fast linear time systolic architecture for computing the greatest common divisor (GCD) of two polynomials is presented. The architecture consists of a number of identical processing elements that accept streams of numbers and perform certain sequential computations on them. All the processing elements are cascaded, so that the output of each one is used as input by the next one. The authors state in the paper that they have not developed a formal proof for their architecture, but rely on "extensive simulation results" to justify it. Here we present a simple proof based on viewing each processing element as a stream transducer. It has been shown by Kahn [4] that if a stream transducer is specified functionally we can use the semantics of applicative languages to reason about it, even though it may be implemented as an imperative program acting on a stream of tokens (as long as the program obeys certain rules). We are thus able to reason about the procedural algorithm of the Kung-Brent processing element using a functional framework.

The rest of this paper is organized as follows. In the following section (Sec II) we briefly describe the architecture of the chip as presented in [1]. Then (in Sec III) we describe how Euclid's algorithm, which forms the basis of the chip's processing can be represented as an algorithm on a new *representation domain* - that of streams of numbers. In Section IV we give a functional interpretation for the action of the processing elements in the Kung-Brent architecture, and decompose the problem of verifying the architecture into three parts - *maintaining the invariant*, *preserving the gcd* and *proof of termination*. Sections V, VI and VII address each of these parts. Finally in Sec VIII we compare our technique with other existing techniques for verifying systolic architectures.

---

[1] Although there exist a large number of systolic arrays that are implemented using a self-timed protocol, conceptually the architecture is viewed as operating synchronously.

II Informal description of the gcd chip

The basis of the gcd-chip is Euclid's algorithm which depends on the following mathematical fact. If A and B are two non-zero polynomials of degrees i and j respectively as shown below,

$$A = a_i x^i + \ldots + a_1 x + a_0$$
$$B = b_j x^j + \ldots + b_1 x + b_0$$

then their gcd satisfies

$$\gcd(A, B) = \begin{cases} \gcd((A - [a_i/b_j]x^{i-j}.B), B) & \text{if } i \geq j \\ \gcd(A, (B - [b_j/a_i]x^{j-i}.A)) & \text{if } j \geq i \end{cases}$$

When $i \geq j$ polynomial A is reduced, and vice versa. This serves as the basis for a gcd-preserving transformation that also reduces the degree of one of the polynomials (by at least one). It is successively applied to the original pair of polynomials until one of them is reduced to the zero polynomial, at which point the other one is the desired gcd. The Kung-Brent architecture consists of i + j processors connected as shown in Figs II-1 and II-2 below.

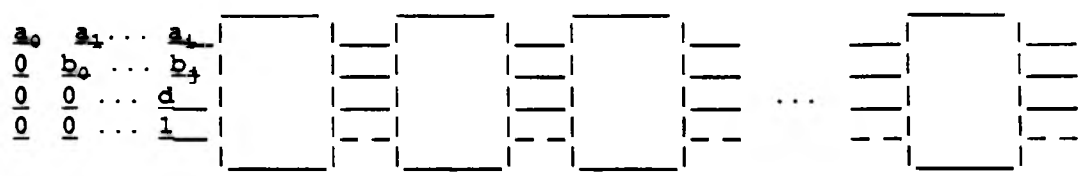

Figure II-1: *Kung-Brent architecture for computing polynomial gcd*

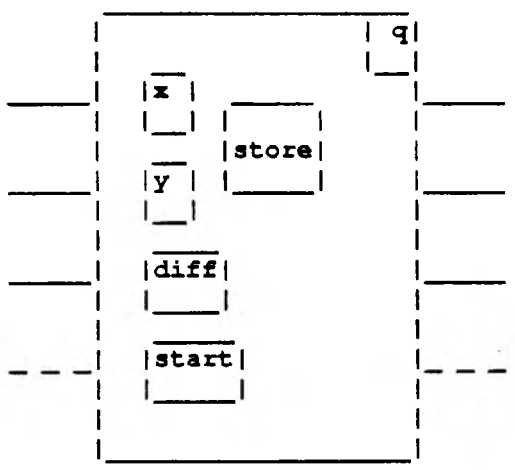

Figure II-2: *Details of a single processor showing registers for the variables*

We express the operation of each processor as a section of imperative *algol style* code as shown in Fig II-3. We have introduced a notation for the input (and output) of values from a particular channel. This notation, namely

"read <a,b,d> into [x,y,diff]" is to be viewed as a parallel assignment of values read from the indicated channels. The write construct should be interpreted analogously. Although this code is not a verbatim copy of the code in [1] it corrects a few minor errors there and expresses the action of the processors as a complete algol-like procedure.

In essence, each processor begins a new cycle when it receives a start bit on the s channel. At this instant it also receives an integer diff on the d channel which represents the difference of the degrees of the two polynomials. Depending on whether d is non-negative of not the processor determines whether stream A or stream B is to be reduced. It then computes the coefficient q, with which the stream to be reduced has to be multiplied, while delaying the other stream by one time unit. After this it cycles through the rest of the input stream (i.e. until a new start bit appears) multiplying one stream by q, and passing the other stream unaltered (except for the delay).

```
input channels a, b, s, d;
output channels ā, b̄, s̄, d̄;
vars q, x, y, start, diff, store;
begin
    repeat read <a, b, d, s> into [x, y, diff, start]
    until start = 1;
    read <a, b, d> into [x, y, diff];
    if diff ≥ 0 then
        begin       (* i.e. reduceA *)
        q := x/y;
        store := y;
        read <a, b, d> into [x, y, diff];
        write [x-q*y, store, 1, diff-1] into <ā, b̄, s̄, d̄>;
        read <s, d> into [start, diff];
        while start ≠ 1 do
            begin
            store := y;
            read <a, b> into [x, y];
            write [x-q*y, store, diff] into <ā, b̄, d̄>;
            read <s, d> into [start, diff];
        end (* while start ≠ 1 *)
    end (* if diff ≥ 0 then *)
    else            (* i.e. reduceB *)
        begin
        q := y/x;
        store := x;
        read <a, b, d> into [x, y, diff];
        write [store, y-q*x, 1, diff+1] into <ā, b̄, s̄, d̄>;
        read <s, d> into [start, diff];
        while start ≠ 1 do
            begin
            store := x;
            read <a, b> into [x, y];
            write [store, y-q*x, diff] into <ā, b̄, d̄>;
            read <s, d> into [start, diff];
        end (* while start ≠ 1 *)
    end (* if)
```

Figure II-3: *Imperative code specifying the behavior of a single processing element*

III Representation of the algorithm on streams of numbers

We think of polynomials as a sequence of coefficient-power pairs, with the powers in descending order, and terms with zero coefficients being explicitly included. The first term must have a nonzero coefficient, and thus its power is the degree of the polynomial. If p_1 and p_2 are two such sequences we can express Euclid's algorithm as follows.

```
gcd(p1, p2) =
    if p1 = [0, 0] then p2
    elseif p2 = [0, 0] then p1
    elseif deg(p1) ≥ deg(p2) then
       gcd((p1 - [Q(p1, p2), (deg(p1) - deg(p2))] * p2), p2)
    else
       gcd(p1, (p2 - [Q(p2, p1), (deg(p2) - deg(p1))] * p1))

where
    deg(p) = second(first(p)) is the degree of p;

    Q(p1, p2) = first(first(p1))/first(first(p2)) is the ratio of the leading
coefficients of p1 and p2; and '-' and '*' are polynomial subtraction and
multiplication respectively.
```

However, the processing elements that perform the gcd-computation do not use this representation of polynomials. The representation used by the algorithm is (roughly speaking) as follows: a polynomial is a stream of numbers, each number being the coefficient of a particular monomial, together with an integer representing the *degree* of the polynomial. The streams may have leading zeroes, and the powers of each monomial are not explicitly indicated, but depend on the position in the stream. The algorithm works directly with *pairs* of polynomials rather than with individual polynomials each represented as indicated above. The input to a single processing element is an integer and a *pair* of streams of numbers. The streams are interpreted as indicated above but the integer encodes information that allows the processing element to determine the *difference* in the degrees of the two polynomials in question. The exact details of the representation are given by the functions defined below.

Since these streams represent polynomials there must exist a representation function *rep* and an associated representation invariant. The representation function *rep* maps pairs of number streams and their associated integer to pairs of polynomials. We define *rep* in terms of another function *prep* which takes an integer d and a single stream of numbers s (which has no leading zeroes) and yields a polynomial as follows.

```
prep(s, d) =
    if d = 0 then
        if s=[] then [0,0] else [first(s), 0]
    elseif s=[] then [0,d]^prep(s, d-1)
    else [first(s), d]^prep(rest(s), d-1)
```

We note here that the Brent-Kung algorithm places a (not too severe) restriction on the polynomials that it can handle. At most one of the polynomials can have a zero constant term. It can be shown that this property is preserved in each recursive invocation of Euclid's algorithm, and hence the gcd polynomial too, has a nonzero constant term. In the following `AllZero(s)` is a predicate asserting that all elements in s are zero; *StripTrailingZeroes*(s) is a function that removes all trailing zeroes from a stream s (provided `AllZero`(s_1) is false); *Final*(i, s) is the stream consisting of the last i elements of a stream s; and ^ represents the infix cons operator. Using *prep* we now define *rep* as follows.

```
    rep(s_A, s_B, d) =
        if AllZero(s_A) then
          [[0,0], prep(StripTrailingZeroes(s_B), length(StripTrailingZeroes(s_B))-1)]
        elseif AllZero(s_B) then
          [prep(StripTrailingZeroes(s_A), length(StripTrailingZeroes(s_A))-1), [0,0]]
        elseif first(s_A)=0 then rep(rest(s_A), s_B, d-1)
        elseif first(s_B)=0 then rep(s_A, rest(s_B), d+1)
        else rep'(StripTrailingZeroes(s_A), StripTrailingZeroes(s_B), d)
where
    rep'(s_1, s_2, d)=
        if d > length(s_1) - length(s_2) then
                [prep(s_1, length(s_2)+d-1), prep(s_2, length(s_2)-1)]
        elseif d = length(s_1) - length(s_2) then
                [prep(s_1, length(s_1)-1), prep(s_2, length(s_1)-1)]
        elseif d < length(s_1) - length(s_2) then
                [prep(s_1, length(s_1)-1), prep(s_2, length(s_1)-d-1)]
```

The representation invariant R is a predicate that indicates the properties that any combination of a pair of streams and an integer must satisfy in order that they can be viable representations of polynomial pairs. This invariant must be preserved by the actions of the processing element. We shall henceforth use R_1, R_2, R_3 and R_4 to denote the four conjuncts of R.

```
        R is defined as
        {  length(s_A) = length(s_B) ∧ length(s_A) > n
         ∧ (first(s_A) ≠ 0 ∨ first(s_B) ≠ 0)
         ∧ d≥0 ⇒ (n≥d ∧ AllZero(Final(d, s_B))) ∨ AllZero(s_B)
         ∧ d<0 ⇒ (n≥-d ∧ AllZero(Final(-d, s_A))) ∨ AllZero(s_A)
        }
```

IV Functional View of the Processing Elements

The action of each processing element is described by a function that we shall henceforth refer to as *reduce*. Since the code in Fig II-3 does not branch on the result of any tests for the *availability of data on the input channels* it satisfies the conditions that were shown by Kahn [4] to guarantee that it can be expressed as a *determinate* function on its input data *streams*. The function corresponding to this code is:

```
reduce(s_A, s_B, d) =
    if first(s_A)=0 or (first(s_B)≠0 ∧ d≥0) then
                [f(rest(s_A), rest(s_B), first(s_A)/first(s_B))^0, s_B, d-1]
    else        [s_A, f(rest(s_B), rest(s_A), first(s_B)/first(s_A))^0, d+1]

where f is defined as
f(s_1, s_2, q) =
    if s_1 = [] then []
    else (first(s_1) - q * first(s_2)) ^ f(rest(s_1), rest(s_2), q)
```

s_A and s_B represent the two streams and d is an integer (we use an integer rather than a stream, since only the first element in the d "stream" is used in the computation). Thus the variables x and y in the code, are *first*(s_A) and *first*(s_B) and q which is used as a storage for an intermediate result is merely passed as a parameter to the function f. The proof of correctness of the action of each processing element then consists of demonstrating that *reduce* satisfies

the following conditions.

- It must preserve the invariant R on every invocation, i.e.

$$R(s_A, s_B, d) \Rightarrow R(reduce(s_A, s_B, d)) \quad (1)$$

- It must preserve the gcd of the polynomials obtained by applying *rep* to its input and output streams, i.e.

$$\gcd[rep(reduce(s_A, s_B, d))] = \gcd[rep(s_A, s_B, d)] \quad (2)$$

- The computation defined by *reduce* "terminates", i.e. if the length of the original streams is m the exactly 2*m-1 applications of *reduce* (denoted by $reduce^{2*m-1}(s_A, s_B, d)$) cause one of the polynomials to become the zero polynomial[1]. This means that

$$first(rep(reduce^{2*m-1}(s_A, s_B, d))) = [0,0]$$
$$\vee\ second(rep(reduce^{2*m-1}(s_A, s_B, d))) = [0,0] \quad (3)$$

In the rest of the paper we shall use f_1 to denote $f(rest(s_A, rest(s_B), first(s_A)/first(s_B)))$ and f_2 to denote $f(rest(s_B, rest(s_A), first(s_B)/first(s_A)))$. We shall now prove each of the above three equations in the following.

V Maintaining the Invariant

To prove Equation 1 above we expand the definition of *reduce* in its right hand side, thus obtaining.

$$R(s_A, s_B, d) \Rightarrow$$
$$[first(s_A)=0 \vee (first(s_B)\neq 0 \wedge d\geq 0) \Rightarrow R(f_1\hat{}0, s_B, d-1)]$$
$$\wedge\ [\sim(first(s_A)=0 \vee (first(s_B)\neq 0 \wedge d\geq 0)) \Rightarrow R(s_A, f_2\hat{}0, d+1)]$$

Substituting the definition of R in all the three occurrences in the above, we get the following.

$$[length(s_A) = length(s_B) = n \wedge n>0 \wedge (first(s_A) \neq 0 \vee first(s_B) \neq 0)$$
$$\wedge\ d\geq 0 \Rightarrow (n\geq d \wedge AllZero(Final(d, s_B))) \vee AllZero(s_B)$$
$$\wedge\ d<0 \Rightarrow (n\geq -d \wedge AllZero(Final(-d, s_A))) \vee AllZero(s_A)]$$

$$\Rightarrow\ [[first(s_A)=0 \vee (first(s_B)\neq 0 \wedge d\geq 0) \Rightarrow$$
$$[length(f_1\hat{}0) = length(s_B) = n \wedge n>0 \wedge (first(f_1\hat{}0) \neq 0 \vee first(s_B) \neq 0)$$
$$\wedge\ d-1\geq 0 \Rightarrow (n\geq d-1 \wedge AllZero(Final(d-1, s_B))) \vee AllZero(s_B)$$
$$\wedge\ d-1<0 \Rightarrow (n\geq -d+1 \wedge AllZero(Final(-d+1, f_1\hat{}0))) \vee AllZero(f_1\hat{}0)]]$$

$$\wedge\ [\sim(first(s_A)=0 \vee (first(s_B)\neq 0 \wedge d\geq 0)) \Rightarrow$$
$$[length(s_A) = length(f_2\hat{}0) = n \wedge n>0 \wedge (first(s_A) \neq 0 \vee first(f_2\hat{}0) \neq 0)$$
$$\wedge\ d+1\geq 0 \Rightarrow (n\geq d+1 \wedge AllZero(Final(d+1, f_2\hat{}0))) \vee AllZero(f_2\hat{}0)$$
$$\wedge\ d+1<0 \Rightarrow (n\geq -d-1 \wedge AllZero(Final(-d-1, s_A))) \vee AllZero(s_A)]]]$$

By a straightforward manipulation of these expressions (see Appendix I) we can show that this is indeed the case. We have thus shown that *reduce* preserves the invariant R.

[1]Note that this formulation of termination is slightly different from the termination in the classical sense. This is because our hardware architecture has a constant number of processors, and thus applies the function *reduce* exactly 2m+1 times, regardless of whether it is redundant or not.

VI Preserving the GCD

We will now prove equation 2 as defined earlier, i.e.

$$gcd[rep(reduce(s_A, s_B, d))] = gcd[rep(s_A, s_B, d)]$$

We see that three cases arise, depending on whether the first elements in the input streams s_A and s_B are zero or not. Note that one of the four combinations, namely $first(s_A)=0 \wedge first(s_B)=0$ is impossible since s_A and s_B satisfy R_1.

Case I: $first(s_A)=0 \wedge first(s_B)\neq 0$ In this case

$$reduce(s_A, s_B, d) = [rest(s_A)\hat{}0, s_B, d-1]$$

and we will show that the polynomials obtained by applying *rep* to the input and output streams are exactly the same.

$$rep(rest(s_A)\hat{}0, s_B, d-1) = rep(s_A, s_B, d)$$

Looking at the RHS and performing case wise analysis depending on the clauses in *rep*, we get the following subcases.

Subcase I.1: $AllZero(s_A)$ (and hence also $AllZero(rest(s_A)\hat{}0)$) Then

$$rep(s_A, s_B, d) = [[0,0], prep(StripTrailingZeroes(s_B), length(StripTrailingZeroes(s_B)))]$$

and the LHS is also

$$rep(rest(s_A)\hat{}0, s_B, d-1) = [[0,0], prep(StripTrailingZeroes(s_B), length(StripTrailingZeroes(s_B)))]$$

Subcase I.2: $AllZero(s_B)$. This is impossible because $first(s_B) \neq 0$.

Subcase I.3: $first(s_A)=0$

Then $rep(s_A, s_B, d)$ reduces to $rep(rest(s_A), s_B, d-1)$. Since here $AllZero(s_A)$ is not true then $AllZero(rest(s_A))$ must also be false, and there is at least one nonzero element in s_A. Let there be k zeroes before the first nonzero element in $rest(s_A)$ (and hence there are k+1 leading zeroes in s_A). This results in the "if $first(s_A)=0$" clause being true on k successive calls to *rep* in the LHS and k+1 calls on the RHS. Thus what we have to prove reduces to

$$rep'(StripTrailingZeroes(rest^k(rest(s_A)\hat{}0)), StripTrailingZeroes(s_B), d-1-k) = rep'(StripTrailingZeroes(rest^{k+1}(s_A)), StripTrailingZeroes(s_B), d-(k+1))$$

i.e.

$$rep'(StripTrailingZeroes(rest^{k+1}(s_A\hat{}0)), StripTrailingZeroes(s_B), d-(k+1)) = rep'(StripTrailingZeroes(rest^{k+1}(s_A)), StripTrailingZeroes(s_B), d-(k+1))$$

which is obviously true because $StripTrailingZeroes(s\hat{}0) = StripTrailingZeroes(s)$ for all s, and in particular for $rest^{k+1}(s_A)$.

Subcase I.4: $first(s_B)=0$ which is impossible.

Subcase I.5: the else clause, which is also impossible, since $first(s_A)$ is zero.

The other two cases, i.e. when $first(s_A)\neq 0 \wedge first(s_B)=0$ and the case when $first(s_A)\neq 0 \wedge first(s_B)\neq 0$ are handled similarly (see Appendix II).

VII Termination

We now prove that the function *reduce* applied on finite streams "terminates". What we really have to show (since the function *reduce* represents processing elements in a systolic array) is a slight variation of conventional termination proofs. We have to show that *exactly* N applications of *reduce* on a pair of finite streams will result in one of the output streams becoming zero, and then by the theorem of the previous section the other stream represents the gcd of the pair of polynomials that the initial pair of streams represented. We also have to show that N is constant, regardless of the degrees of the initial polynomials. In particular we will show that if m is the length of the input streams, then exactly 2*m-1 applications of reduce suffice, i.e.

$$reduce^{2*m-1}(s_A, s_B, d) = [p_1, p_2]$$

where **either** $p_1 = [0,0]$ **or** $p_2 = [0,0]$

To prove this we first see that if at any time $first(rep(s_A,s_B,d))$ becomes the zero polynomial [0,0] then applying *reduce* any number of times has no effect on the polynomials represented, i.e.

$$first(rep(s_A, s_B, d)) = [0,0] \Rightarrow$$
$$first(rep(reduce(s_A, s_B, d))) = [0,0]$$
$$\wedge\ second(rep(reduce(s_A, s_B, d))) = second(rep(s_A, s_B, d))$$

and similarly

$$second(rep(s_A, s_B, d)) = [0,0] \Rightarrow$$
$$second(rep(reduce(s_A, s_B, d))) = [0,0]$$
$$\wedge\ first(rep(reduce(s_A, s_B, d))) = first(rep(s_A, s_B, d))$$

We now define a measure function N and show that it is reduced by 1 on every call to *reduce*. N is defined as

$$N(s_A, s_B, d) = d_1 + d_2 + k$$

where
$d_1 = deg(first(rep(s_A, s_B, d)))$ **is the degree of the first polynomial,**
$d_2 = deg(second(rep(s_A, s_B, d)))$ **is the degree of the second polynomial;**
and
k is the number of leading zeroes in one of the streams (remember that both streams cannot simultaneously have leading zeroes).

Note that N is undefined if $AllZero(s_A)$ **or** $AllZero(s_B)$ **is true.**

We will now show that

$$N(s_A, s_B, d) = N(reduce(s_A, s_B, d)) + 1$$
or
$$first(rep(reduce(s_A, s_B, d))) = [0,0]$$
or
$$second(rep(reduce(s_A, s_B, d))) = [0,0] \qquad (4)$$

As before we have three cases:

Case I: $first(s_A) = 0 \wedge first(s_B) \neq 0$. In this case

$$reduce(s_A, s_B, d) = [rest(s_A)\hat{}0, s_B, d-1]$$

It has been shown in the previous section that the polynomials (and hence the sum of their degrees) that are represented by the streams on the rhs of this equation are exactly the same as $rep(s_A, s_B, d)$. Since the number of leading zeroes in s_A is exactly one more than the number of leading zeroes in $rest(s_A)$, Eqn 4 holds.

Case II: $first(s_A) \neq 0 \wedge first(s_B) = 0$. This is exactly analogous to the above case.

Case III: $first(s_A) \neq 0 \wedge first(s_B) \neq 0$. Let us assume that $d \geq 0$. The $d < 0$ case can be proved in a similar manner. We then have

$$reduce(s_A, s_B, d) = [f_1\hat{}0, s_B, d-1]$$

As before we have the following subcases depending on the clauses in $rep(f_1\hat{}0, s_B, d-1)$.

Subcase III.1 $AllZero(f_1\hat{}0)$. In this case we are done, since

$$first(rep(reduce(s_A, s_B, d))) = [0,0]$$

Subcase III.2 $first(f_1\hat{}0) = 0$. As before, let us assume that there are exactly k leading zeroes in f_1 (since $AllZero(f_1\hat{}0)$ is false). As has been shown earlier, this means that

$$rep(f_1\hat{}0, s_B, d-1) = rep'(StripTrailingZeroes(rest^k(f_1\hat{}0)), StripTrailingZeroes(s_B), d-k-1)$$

and d - k - 1 is the difference of the degrees of the two polynomials. Thus we have the following

$$deg(first(rep(f_1\hat{}0, s_B, d-1))) = deg(second(rep(s_A, s_B, d))) + d-1-k$$

We also know that since $first(s_A) \neq 0$ and $first(s_B) \neq 0$

$$deg(first(rep(s_A, s_B, d))) = deg(second(rep(s_A, s_B, d))) + d$$

And it has been shown in the previous subsection that for this case (i.e. $d \geq 0$)

$$second(rep(s_A, s_B, d)) = second(rep(reduce(s_A, s_B, d)))$$

Using the above then, we have

$$\begin{aligned} &N(reduce(s_A, s_B, d)) = \\ &\quad deg(first(rep(f_1\hat{}0, s_B, d-1))) + deg(second(rep(f_1\hat{}0, s_B, d-1))) \\ &\qquad + \text{leading zeroes in } f_1\hat{}0 \\ &\quad = deg(first(rep(f_1\hat{}0, s_B, d-1))) + deg(second(rep(f_1\hat{}0, s_B, d-1))) + k \\ &\quad = 2 * deg(second(rep(s_A, s_B, d))) + d - 1 - k + k \\ &\quad = N(s_A, s_B, d) - 1 \end{aligned}$$

Subcase III.3: $AllZero(s_B)$. This is impossible since $first(s_B) \neq 0$.

Subcase III.4: $first(s_B) = 0$ Again this is impossible.

Subcase III.5: the else clause. The reasoning for this is identical to subcase III.2 with k being zero.

We have thus shown that $N(s_A, s_B, d)$ either decreases by exactly one on every call to *reduce*, or that one of the streams consists of only zeroes. In the worst case then, the initial value of $N(s_A, s_B, d)$ is $2 * (m-1)$ and it will take $2*m - 2$ calls of *reduce* to make $N(s_A, s_B, d)$ equal to 0. But this means that both polynomials have degree zero and there are no leading zeroes in either stream (since each of these three values is non-negative and their sum is zero). The very next call on *reduce* will then make the stream s_A all zeroes and leave s_B unchanged. At this point the gcd of the two polynomials is *first*(s_B) and we are done, since the gcd of two polynomials of degree zero is (any) polynomial of degree zero, and [*first*(s_B), 0] in particular. Thus at most $2*m - 1$ calls on *reduce* makes one of the polynomials [0, 0]

VIII Conclusions

The specific example that we have proved here illustrates the following general technique for the verification of systolic architectures. The action of the individual processing elements of an array are modelled as stream-functions by extending their token-wise actions (which are not usually functional). Expressing the array connectivity is then trivial - it merely corresponds to function composition. The action of the entire array is obtained by setting up a set of mutually recursive equations between these functions and solving them via fixed-point theory.

This approach is similar to that presented in Chen's thesis ([2]) where space-time recursion equations are set up and their least fixed point yields the semantic function for the entire network [7]. An important distinction in our approach is that by first expressing the functionality of each processing element as a stream transducer we have abstracted away the time component. Successive time instants (which need to be described explicitly in Chen's approach) are represented by successive appearances of tokens on a stream. Since the theory of functions and function-composition is fairly well understood, this gives us additional leverage when manipulating our expressions.

Melhem and Rheinboldt [5] describe a formalism involving graphs with colored arcs in order to reason about systolic networks. The framework presented there is very similar to ours. The processing elements are also modelled as (essentially) stream functions, though they are referred to as operators. The color on the arcs indicates the connectivity. The overall network behavior is captured by a system of equations, *difference equations* rather than our recursive equations, and network behavior is obtained by solving these equations. Purushothaman (in [6]) also has a similar approach, where the verifivation problem is also reduced to the solution of recurrence equations. In both these approaches the equations are set up usning the token-wise behavior of the individual processing elements, and standard techniques for manipulating functions are not exploited. Their use of difference/recurrence equations implies that both, the data representation and processor behavior issues have to be handled simultaneously. These issues are cleanly separated in our approach.

IX References

1. Brent, R. P. and Kung, H. T. Systolic VLSI Arrays for Linear-Time GCD Computation. VLSI 83, Aug, 1983, pp. 145:154.

2. Chen, Marina C. *Space-Time Algorithms: Semantics and Methodology.* Ph.D. Th., California Institute of Technology, Pasadena, CA, May 1983.

3. Hoare, C. A. R. "Proof of Correctness of Data Representations". *Acta Informatica 1* (1972), 271-281.

4. Kahn, Gilles. The Semantics of a Simple Language for Parallel Processing. Proceedings of IFIP, IFIP, Aug, 1974, pp. 471-475.

5. Melhem, Rami G. and Rheinboldt, Werner C. "A Mathematical Model for the Verification of Systolic Networks". *SIAM Journal of Computing 13*, 3 (August 1984), 541-565.

6. Purushothaman, S. *Verification of Systolic Architectures.* Ph.D. Th., University of Utah, Salt Lake City, Utah 84112, December 1985.

7. Joseph Stoy. *Denotational Semantics: The Scott-Stratchey Approach to Programming Language Theory.* MIT Press, 1977.

Appendix I Proof of Invariant Preservation

We shall now prove the theorem stated in Section V, i.e.

$$
\begin{aligned}
&[\mathit{length}(\mathbf{s_A}) = \mathit{length}(\mathbf{s_B}) = \mathbf{n} \wedge \mathbf{n}>0 \wedge (\mathit{first}(\mathbf{s_A}) \neq 0 \vee \mathit{first}(\mathbf{s_B}) \neq 0) \\
&\wedge\ \mathbf{d}\geq 0 \Rightarrow (\mathbf{n}\geq \mathbf{d} \wedge \mathbf{AllZero}(\mathit{Final}(\mathbf{d}, \mathbf{s_B}))) \vee \mathbf{AllZero}(\mathbf{s_B}) \\
&\wedge\ \mathbf{d}<0 \Rightarrow (\mathbf{n}\geq -\mathbf{d} \wedge \mathbf{AllZero}(\mathit{Final}(-\mathbf{d}, \mathbf{s_A}))) \vee \mathbf{AllZero}(\mathbf{s_A})] \\
&\Rightarrow [[\mathit{first}(\mathbf{s_A}) = 0 \vee (\mathit{first}(\mathbf{s_B}) \neq 0 \wedge \mathbf{d}\geq 0) \Rightarrow \\
&\qquad [\mathit{length}(f_1\hat{}0) = \mathit{length}(\mathbf{s_B}) = \mathbf{n} \wedge \mathbf{n}>0 \wedge (\mathit{first}(f_1\hat{}0) \neq 0 \vee \mathit{first}(\mathbf{s_B}) \neq 0) \\
&\qquad\ \wedge\ \mathbf{d}-1\geq 0 \Rightarrow (\mathbf{n}\geq \mathbf{d}-1 \wedge \mathbf{AllZero}(\mathit{Final}(\mathbf{d}-1, \mathbf{s_B}))) \vee \mathbf{AllZero}(\mathbf{s_B}) \\
&\qquad\ \wedge\ \mathbf{d}-1<0 \Rightarrow (\mathbf{n}\geq -\mathbf{d}+1 \wedge \mathbf{AllZero}(\mathit{Final}(-\mathbf{d}+1, f_1\hat{}0))) \vee \mathbf{AllZero}(f_1\hat{}0)]] \\
&\wedge\ [\sim(\mathit{first}(\mathbf{s_A}) = 0 \vee (\mathit{first}(\mathbf{s_B}) \neq 0 \wedge \mathbf{d}\geq 0)) \Rightarrow \\
&\qquad [\mathit{length}(\mathbf{s_A}) = \mathit{length}(f_2\hat{}0) = \mathbf{n} \wedge \mathbf{n}>0 \wedge (\mathit{first}(\mathbf{s_A}) \neq 0 \vee \mathit{first}(f_2\hat{}0) \neq 0) \\
&\qquad\ \wedge\ \mathbf{d}+1\geq 0 \Rightarrow (\mathbf{n}\geq \mathbf{d}+1 \wedge \mathbf{AllZero}(\mathit{Final}(\mathbf{d}+1, f_2\hat{}0))) \vee \mathbf{AllZero}(f_2\hat{}0) \\
&\qquad\ \wedge\ \mathbf{d}+1<0 \Rightarrow (\mathbf{n}\geq -\mathbf{d}-1 \wedge \mathbf{AllZero}(\mathit{Final}(-\mathbf{d}-1, \mathbf{s_A}))) \vee \mathbf{AllZero}(\mathbf{s_A})]]]
\end{aligned}
$$

To prove this we first note that [*length*(s)>0] ⇒ [*length*(s) = *length*(*rest*(s))+1]; and we can easily show from the definition of *f* that

$$
\begin{aligned}
&\mathit{length}(\mathbf{s_1}) = \mathit{length}(\mathbf{s_2}) \wedge \mathit{length}(\mathbf{s_1})>0 \wedge \mathit{first}(\mathbf{s_2}) \neq 0 \Rightarrow \\
&\qquad\qquad \mathit{length}(f(\mathbf{s_1}, \mathbf{s_2}, \mathbf{q})) = \mathit{length}(\mathbf{s_2}) \\
&\Rightarrow \mathit{length}(f(\mathit{rest}(\mathbf{s_1}), \mathit{rest}(\mathbf{s_2}), \mathit{first}(\mathbf{s_1})/\mathit{first}(\mathbf{s_2}))\hat{}0) = \mathit{length}(\mathit{rest}(\mathbf{s_2})) + 1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad = \mathit{length}(\mathbf{s_2})
\end{aligned}
$$

And similarly

$$
\begin{aligned}
&\mathit{length}(\mathbf{s_1}) = \mathit{length}(\mathbf{s_2}) \wedge \mathit{length}(\mathbf{s_1})>0 \wedge \mathit{first}(\mathbf{s_1}) \neq 0 \Rightarrow \\
&\mathit{length}(f(\mathit{rest}(\mathbf{s_2}), \mathit{rest}(\mathbf{s_1}), \mathit{first}(\mathbf{s_2})/\mathit{first}(\mathbf{s_1}))\hat{}0) = \mathbf{i(length)}(\mathit{rest}(\mathbf{s_1})) + 1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad = \mathit{length}(\mathbf{s_1})
\end{aligned}
$$

Also we see that

$$
\begin{aligned}
&(\mathit{first}(\mathbf{s_A}) \neq 0 \vee \mathit{first}(\mathbf{s_B}) \neq 0) \Rightarrow \\
&\qquad [\mathit{first}(\mathbf{s_A}) = 0 \vee (\mathit{first}(\mathbf{s_B}) \neq 0 \wedge \mathbf{d}\geq 0) \Rightarrow \mathit{first}(\mathbf{s_B}) \neq 0]
\end{aligned}
$$

and also

$$
\begin{aligned}
&(\mathit{first}(\mathbf{s_A}) \neq 0 \vee \mathit{first}(\mathbf{s_B}) \neq 0) \Rightarrow \\
&\qquad [\sim(\mathit{first}(\mathbf{s_A}) = 0 \vee (\mathit{first}(\mathbf{s_B}) \neq 0 \wedge \mathbf{d}\geq 0)) \Rightarrow \mathit{first}(\mathbf{s_A}) \neq 0]
\end{aligned}
$$

Thus, by combining the above four implications we get

$$
\begin{aligned}
&\mathit{length}(\mathbf{s_A}) = \mathit{length}(\mathbf{s_B}) = \mathbf{n} \wedge \mathbf{n}>0 \wedge (\mathit{first}(\mathbf{s_A}) \neq 0 \vee \mathit{first}(\mathbf{s_B}) \neq 0) \Rightarrow \\
&\quad [(\mathit{first}(\mathbf{s_A}) = 0 \vee (\mathit{first}(\mathbf{s_B}) \neq 0 \wedge \mathbf{d} \geq 0)) \Rightarrow \\
&\qquad (\mathit{length}(f_1\hat{}0) = \mathit{length}(\mathbf{s_B}) = \mathbf{n} \wedge \mathbf{n}>0) \wedge (\mathit{first}(f_1\hat{}0) \neq 0 \vee \mathit{first}(\mathbf{s_B}) \neq 0)] \\
&\wedge [\sim(\mathit{first}(\mathbf{s_A}) = 0 \vee (\mathit{first}(\mathbf{s_B}) \neq 0 \wedge \mathbf{d} \geq 0)) \Rightarrow \\
&\qquad (\mathit{length}(\mathbf{s_A}) = \mathit{length}(f_2\hat{}0) = \mathbf{n} \wedge \mathbf{n}>0) \wedge (\mathit{first}(\mathbf{s_A}) \neq 0 \vee \mathit{first}(f_2\hat{}0) \neq 0)]
\end{aligned}
$$

i.e. we have shown that (note that R_1, R_2, R_3 and R_4 are the conjuncts of R)

$$R_1(s_A, s_B, d) \wedge R_2(s_A, s_B, d) \Rightarrow R_1(\mathit{reduce}(s_A, s_B, d)) \wedge R_2(\mathit{reduce}(s_A, s_B, d)).$$

We will now show that

$$R_3(s_A, s_B, d) \wedge R_4(s_A, s_B, d) \Rightarrow R_3(\mathit{reduce}(s_A, s_B, d)) \wedge R_4(\mathit{reduce}(s_A, s_B, d))$$

i.e.

$$
\begin{aligned}
&[\mathbf{d}\geq 0 \Rightarrow (\mathbf{n}\geq \mathbf{d} \wedge \mathbf{AllZero}(\mathit{Final}(\mathbf{d}, \mathbf{s_B}))) \vee \mathbf{AllZero}(\mathbf{s_B}) \\
&\wedge \mathbf{d}<0 \Rightarrow (\mathbf{n}\geq -\mathbf{d} \wedge \mathbf{AllZero}(\mathit{Final}(-\mathbf{d}, \mathbf{s_A}))) \vee \mathbf{AllZero}(\mathbf{s_A})] \Rightarrow \\
&\quad [[\mathit{first}(\mathbf{s_A}) = 0 \vee (\mathit{first}(\mathbf{s_B}) \neq 0 \wedge \mathbf{d}\geq 0) \Rightarrow \\
&\qquad [\mathbf{d}-1\geq 0 \Rightarrow (\mathbf{n}\geq \mathbf{d}-1 \wedge \mathbf{AllZero}(\mathit{Final}(\mathbf{d}-1, \mathbf{s_B}))) \vee \mathbf{AllZero}(\mathbf{s_B}) \\
&\qquad \wedge \mathbf{d}-1<0 \Rightarrow (\mathbf{n}\geq -\mathbf{d}+1 \wedge \mathbf{AllZero}(\mathit{Final}(-\mathbf{d}+1, f_1\hat{}0))) \vee \mathbf{AllZero}(f_1\hat{}0)] \\
&\quad] \\
&\wedge [\sim(\mathit{first}(\mathbf{s_A}) = 0 \vee (\mathit{first}(\mathbf{s_B}) \neq 0 \wedge \mathbf{d}\geq 0)) \Rightarrow \\
&\qquad [\mathbf{d}+1\geq 0 \Rightarrow (\mathbf{n}\geq \mathbf{d}+1 \wedge \mathbf{AllZero}(\mathit{Final}(\mathbf{d}+1, f_2\hat{}0))) \vee \mathbf{AllZero}(f_2\hat{}0) \\
&\qquad \wedge \mathbf{d}+1<0 \Rightarrow (\mathbf{n}\geq -\mathbf{d}-1 \wedge \mathbf{AllZero}(\mathit{Final}(-\mathbf{d}-1, \mathbf{s_A}))) \vee \mathbf{AllZero}(\mathbf{s_A})]]]
\end{aligned}
\qquad (5)
$$

This is shown by means of a case by case analysis on the value of d.

Case I: d > 0 In this case equation 5 reduces to

$$
\begin{aligned}
&[(\mathbf{n}\geq \mathbf{d} \wedge \mathbf{AllZero}(\mathit{Final}(\mathbf{d}, \mathbf{s_B}))) \vee \mathbf{AllZero}(\mathbf{s_B})] \Rightarrow \\
&\quad (\mathit{first}(\mathbf{s_A}) = 0 \vee \mathit{first}(\mathbf{s_B}) \neq 0) \Rightarrow \\
&\qquad (\mathbf{n}\geq \mathbf{d}-1 \wedge \mathbf{AllZero}(\mathit{Final}(\mathbf{d}-1, \mathbf{s_B}))) \vee \mathbf{AllZero}(\mathbf{s_B}) \\
&\wedge (\mathit{first}(\mathbf{s_A}) \neq 0 \wedge \mathit{first}(\mathbf{s_B}) = 0) \Rightarrow \\
&\qquad (\mathbf{n}\geq \mathbf{d}+1 \wedge \mathbf{AllZero}(\mathit{Final}(\mathbf{d}+1, f_2\hat{}0))) \vee \mathbf{AllZero}(f_1\hat{}0)
\end{aligned}
$$

The first implication in the RHS of the above is obviously true since

$$\mathbf{n} \geq \mathbf{d} \Rightarrow \mathbf{n}\geq \mathbf{d}-1$$

$$\mathbf{d} > 0 \wedge \mathbf{AllZero}(\mathit{Final}(\mathbf{d}, \mathbf{s_B})) \Rightarrow \mathbf{AllZero}(\mathit{Final}(\mathbf{d}-1, \mathbf{s_B}))$$

and $\quad \mathbf{AllZero}(\mathbf{s_B}) \Rightarrow \mathbf{AllZero}(\mathbf{s_B})$

Also we know that

$$\mathit{first}(\mathbf{s_B}) = 0 \Rightarrow f_2\hat{}0 = \mathit{rest}(\mathbf{s_B})\hat{}0$$

and

$$\mathbf{n}=\mathbf{d} \Rightarrow \mathbf{AllZero}(\mathit{Final}(\mathbf{d}, \mathbf{s_B}))$$

Hence we have two subcases: d=n and d≠n. In the former case the implication reduces to

$$\mathbf{AllZero}(\mathbf{s_B}) \Rightarrow \mathbf{AllZero}(\mathit{rest}(\mathbf{s_B})\hat{}0)$$

which is obviously true, and the latter case (n≠d) implies that

$n \geq d \Rightarrow n \geq d+1$

. This reduces the final implication to

$\mathbf{AllZero}(\mathit{Final}(d, s_B)) \Rightarrow \mathbf{AllZero}(\mathit{Final}(d+1, \mathit{rest}(s_B)\hat{}0)) \vee \mathbf{AllZero}(\mathit{rest}(s_B)\hat{}0)$

which is again obviously true.

Case II: d = 0. Here the implication reduces to

$$\mathit{first}(s_A)=0 \vee \mathit{first}(s_B)\neq 0 \Rightarrow n \geq 1 \wedge \mathbf{AllZero}(\mathit{Final}(1, f_1\hat{}0)) \vee \mathbf{AllZero}(f_1\hat{}0)$$
$$\wedge \ \mathit{first}(s_A)\neq 0 \wedge \mathit{first}(s_B)=0 \Rightarrow n \geq 1 \wedge \mathbf{AllZero}(\mathit{Final}(1, f_2\hat{}0)) \vee \mathbf{AllZero}(f_2\hat{}0)$$

which is again true since $n \geq 1$ follows from $n>0$ which is part of the initial invariant, and AllZero(*Final*(1, s^0)) is always true for all streams s.

Case III: d = -1. In this case we have to show that

$$(n \geq 1 \wedge \mathbf{AllZero}(\mathit{Final}(1, s_A))) \vee \mathbf{AllZero}(s_A) \Rightarrow$$
$$\mathit{first}(s_A)=0 \Rightarrow (n \geq 2 \wedge \mathbf{AllZero}(\mathit{Final}(2, \mathit{rest}(s_A)\hat{}0))) \vee \mathbf{AllZero}(\mathit{rest}(s_A)\hat{}0)$$
$$\wedge \ \mathit{first}(s_A)\neq 0 \Rightarrow$$
$$(n \geq 0 \wedge \mathbf{AllZero}(\mathit{Final}(0, f_2\hat{}0))) \vee \mathbf{AllZero}(f_2)\hat{}0)$$
$$\wedge \ (n \geq 0 \wedge \mathbf{AllZero}(\mathit{Final}(0, s_A))) \vee \mathbf{AllZero}(s_A)$$

The first implication is true because n=1 implies that *rest*(s_A) is nil, thus making AllZero(*rest*(s_A)^0) true, while n>1 and the LHS directly imply the RHS of the first implication. The second implication is also trivially true since $n \geq 0$ and AllZero(*Final*(0, s)) is always true.

Case IV: d<-1. In this case we have to prove that

$$(n \geq -d \wedge \mathbf{AllZero}(\mathit{Final}(-d, s_A))) \vee \mathbf{AllZero}(s_A) \Rightarrow$$
$$\mathit{first}(s_A)=0 \Rightarrow (n \geq -d+1 \wedge \mathbf{AllZero}(\mathit{Final}(-d+1, f_1\hat{}0))) \vee \mathbf{AllZero}(f_1\hat{}0)$$
$$\wedge \ \mathit{first}(s_A)\neq 0 \Rightarrow (n \geq -d-1 \wedge \mathbf{AllZero}(\mathit{Final}(-d-1, s_A))) \vee \mathbf{AllZero}(s_A)$$

The first implication is proved in a similar manner to Case III (but with n=d and n>d as the two subcases). The second on is also trivially true, by an argument similar to Case I.

Appendix II Proof of gcd-preservation - Cases II and III

We now prove that in the two remaining cases described in Section VI the function *reduce* preserves the gcd of the polynomials represented by the inputs.

Case II: *first*(s_A)≠0 ∧ *first*(s_B)=0.

The proof is exactly analogous to Case I, with the same arguments used on s_A instead of s_B. In both these cases the polynomials represented by the streams remain unchanged by the action of *reduce*. It is the third case discussed below that performs the actual reduction.

Case III: *first*(s_A)≠0 ∧ *first*(s_B)≠0.

At this point the first four clauses in the body of *rep*(s_A, s_B, d) are false, and hence d is the difference between the degrees of the two polynomials. We thus have two further subcases - $d \geq 0$ and d<0 (the if clause in the body of

reduce), which correspond to the "if $\deg(p_1)\geq\deg(p_2)$" clause in the body of *gcd*. We will give the detailed proof only for the former case; the d<0 case is proved analogously. Now, *reduce*(s_A, s_B, d) returns (f_1^0, s_B, d-1), and we will show that the polynomials obtained by applying *rep* to the streams are those returned by one recursion of the function *gcd*, i.e.

$$rep(f_1\text{^}0,\ s_B,\ d-1) = [p_1 - p_2 * [Q(p_1,p_2),\ \deg(p_1)-\deg(p_2)],\ p_2], \qquad (6)$$

where

$p_1 = first(rep(s_A,\ s_B,\ d))$, and $p_2 = second(rep(s_A,\ s_B,\ d))$,

Now, we know that $Q(p_1,p_2)$ is $first(first(p_1))/first(first(p_2))$ i.e. $first(s_A)/first(s_B)$ (let's call it q) and $\deg(p_1)-\deg(p_2)$ is d. We also know that since $first(s_A)\neq 0 \wedge first(s_B)\neq 0$

$$rep(s_A,\ s_B,\ d) = rep'(StripTrailingZeroes(s_A),\ StripTrailingZeroes(s_B),\ d)$$

Let $length(StripTrailingZeroes(s_A))$ be l_A, $length(StripTrailingZeroes(s_B))$ be l_B, and $l_A - l_B$ be l. We thus have the following two subcases (depending on whether d≥l or d<l)[1]

Subcase III.1: d≥l. Then

$p_1 = prep(StripTrailingZeroes(s_A),\ l_B+d-1)$

and

$p_2 = prep(StripTrailingZeroes(s_B),\ l_B-1)$

We now have five further subcases depending on the clauses of *rep* in the LHS of Eqn 6.

Subcase III.1.1: $AllZero(f_1\text{^}0)$, i.e. $AllZero(f_1)$. Then

$first(rep(f_1\text{^}0,\ s_B,\ d-1)) = [0,\ 0]$

and

$second(rep(f_1\text{^}0,\ s_B,\ d-1)) = prep(StripTrailingZeroes(s_B),\ length(StripTrailingZeroes(s_B))) = p_2$

Now, from the definition of *f*,

$$AllZero(f(s_1,\ s_2,\ q)) \Rightarrow \forall i\ s_1[i] = q * s_2[i]$$

and thus

$$AllZero(f(rest(s_A),\ rest(s_B),\ q)) \Rightarrow \forall i\ rest(s_A)[i] = q * rest(s_B)[i]$$

Also, since $q = first(s_A) / first(s_B)$ we have

$\forall i\ s_A[i] = q * s_B[i]$, and thus by *lemma* L1[2] this yields

$first(rep(s_A, s_B, d)) = [q,\ d] * second(rep(s_A, s_B, d))$, i.e.

---

[1] in proving the case when d<0 the two subcases would be d>l and d≤l.

[2] Lemma L1 is:

if $\forall i\ s_A[i] = q * s_B[i] \wedge q\neq 0$ then

$first(rep(s_A, s_B, d)) = [q, d] * second(rep(s_A, s_B, d))$.

$p_1 = [q,\ d] * p_2$, which directly implies (6).

Subcase III.1.2 *AllZero*(s_B)

This is impossible since *first*(s_B) ≠ 0.

Subcase III.1.3 *first*(f_1^0) = 0

As before, we assume without loss of generality that there are exactly k leading zeroes in f_1(and hence in f_1^0, since *AllZero*(f_1^0) is false). Then

$$rep\,(f_1\text{^}0,\ s_B,\ d\text{-}1) = rep'\,(StripTrailingZeroes\,(rest^k\,(f_1\text{^}0)),\ StripTrailingZeroes\,(s_B),\ d\text{-}k\text{-}1)$$

and thus d-k-1 is the difference between the degrees of the two polynomials[1]. We now claim that the second polynomial represented by the output streams still does not have a zero constant term, i.e.

$$d\text{-}1\text{-}k \geq length\,(StripTrailingZeroes\,(f_1\text{^}0)) - l_B$$

To prove this we first see from the definition of *StripTrailingZeroes* that for $0<j\leq n$ $s_A[l_A+j] = 0$. Since $l_B + 1 = l_A$ and $d \geq 1$ this means that

$s_A[l_B+d+j] = 0$ for $n-l_B-d \geq j > 0$

and since $d\geq 0$ $s_B[l_B+d+j] = 0\ \forall n-l_B-d\geq j>0$.

Hence

$$rest\,(s_A)\,[l_B+d+j-1] = 0 \qquad j>0$$
$$\text{and} \quad rest\,(s_B)\,[l_B+d+j-1] = 0 \qquad j>0$$

Thus by the fact that $s1[i]=0 \wedge s2[i] = 0 \Rightarrow f(s1,\ s2,\ q)\,[i]=0$,

$$f_1[l_B+d+j-1] = 0 \qquad j>0$$

Hence *length*(*StripTrailingZeroes*(f_1)) ≤ l_B+d-1.

Rearranging the terms and using the fact that

length(*StripTrailingZeroes*(*rest*^{k}(f_1^0))) = *length*(*StripTrailingZeroes*(f_1)) - k

we prove our claim. Thus

$$first\,(rep\,(f_1\text{^}0,\ s_B,\ d\text{-}1)) = prep\,(StripTrailingZeroes\,(rest^k\,(f_1)),\ l_B + (d\text{-}1\text{-}k) - k)$$
$$\text{and}$$
$$second\,(rep\,(f_1\text{^}0,\ s_B,\ d\text{-}1)) = p_2$$

Thus, using lemmas 2 and 3 we can say that

---

[1] This can be readily seen by comparing the second arguments to the two calls on *prep* in each of the three clauses of *rep'*.

prep(*StripTrailingZeroes*($rest^k(f_1)$), l_B + (d-1-k) - k) is the same as p_1 - [q, d] * p_2.

Subcase III.1.4 *first*(s_B) = 0 which is impossible.

Subcase III.1.5 The else clause. This is merely a special case of **Subcase III.1.3** where the value of k is 0.

Subcase III.2: d < l. Then

Let d = l + m, where m > 0. We also have

$$p_1 = prep(StripTrailingZeroes(s_A),\ l_A\text{-}1)$$
$$\text{and} \quad p_2 = prep(StripTrailingZeroes(s_B),\ l_A\text{-}d\text{-}1)$$

As before we have five subcases (depending on the clauses of *rep* in the LHS of Eqn 5).

Subcase III.2.1 *AllZero*(f_1^0). We shall prove that this is impossible, as follows.

Since l > 0 we see that $s_A[l_A] \neq 0$ and $s_B[l_A] = 0$. Now since,

$$\begin{aligned} f_1[l_A\text{-}1] &= rest(s_A)[l_A\text{-}1] - q*rest(s_B)[l_A\text{-}1] \\ &= s_A[l_A] - q*s_B[l_A] \\ &\neq 0 \end{aligned}$$

Hence *AllZero*(f_1) is false which directly implies that *AllZero*(f_1^0).

Subcase III.2.2 *AllZero*(s_B). This is again impossible since *first*(s_B)≠0.

Subcase III.2.3. *first*(f_1^0) = 0. As before, we assume that there are exactly k leading zeroes in f_1, and as before we see that

$$rep(f_1\text{^}0,\ s_B,\ d\text{-}1) = rep'(StripTrailingZeroes(rest^k(f_1\text{^}0)),\ StripTrailingZeroes(s_B),\ d\text{-}k\text{-}1)$$

We now claim that the resultant first polynomial still has a nonzero constant term, and the second one doesn't, i.e.

$$d\text{-}1\text{-}k < length(StripTrailingZeroes(rest^k(f_1\text{^}0))) - l_B$$
$$\text{i.e.} \quad d \leq length(StripTrailingZeroes(f_1)) - l_B$$

To prove it we note that as shown above $f_1[l_A\text{-}1] \neq 0$, which implies that

$$length(StripTrailingZeroes(f_1)) \geq l_A$$
$$\text{i.e.} \quad length(StripTrailingZeroes(f_1)) \geq l_B + 1$$

Since l > d this directly implies our claim above. Again, by lemmas 2 and 3 we see that

prep(*StripTrailingZeroes*($rest^k(f_1)$), l_B + (d-1-k) - k) is the same as p_1 - [q, d] * p_2.

Subcase III.2.4 $first(s_B) = 0$ which is impossible.

Subcase III.2.5 the else clause. Again this is a particular case of Subcase III.2.3 above with k being 0.